LEARNING TO REPRESENT EDITS

Anonymous authors
Paper under double-blind review

ABSTRACT

We introduce the problem of learning distributed representations of edits. By combining a "neural editor" with an "edit encoder", our models learn to represent the salient information of an edit and can be used to apply edits to new inputs. We experiment on natural language and source code edit data. Our evaluation yields promising results that suggest that our neural network models learn to capture the structure and semantics of edits. We hope that this interesting task and data source will inspire other researchers to work further on this problem.

1 INTRODUCTION

One great advantage of electronic storage of documents is the ease with which we can edit them, and edits are performed in a wide variety of contents. For example, right before a conference deadline, papers worldwide are finalized and polished, often involving common fixes for grammar, clarity and style. Would it be possible to automatically extract rules from these common edits? Similarly, program source code is constantly changed to implement new features, follow best practices and fix bugs. With the widespread deployment of (implicit) version control systems, these edits are quickly archived, creating a major data stream that we can learn from.

In this work, we study the problem of learning distributed representations of edits. We only look at small edits with simple semantics that are more likely to appear often and do not consider larger edits; *i.e.*, we consider "add definite articles" rather than "rewrite act 2, scene 3." Concretely, we focus on two questions: i) Can we group semantically equivalent edits together, so that we can automatically recognize common edit patterns? ii) Can we automatically transfer edits from one context to another? A solution to the first question would yield a practical tool for copy editors and programmers alike, automatically identifying the most common changes. By leveraging tools from program synthesis, such groups of edits could be turned into interpretable rules and scripts (Rolim et al., 2017). When there is no simple hard rule explaining how to apply an edit, an answer to the second question would be of great use, *e.g.*, to automatically rewrite natural language following some stylistic rule.

We propose to handle edit data in an autoencoder-style framework, in which an "edit encoder" f_Δ is trained to compute a representation of an edit $\boldsymbol{x}_- \rightarrow \boldsymbol{x}_+$, and a "neural editor" α is trained to construct $\boldsymbol{x}_+$ from the edit representation and $\boldsymbol{x}_-$. This framework ensures that the edit representation is semantically meaningful, and a sufficiently strong neural editor allows this representation to not be specific to the changed element. We experiment with various neural architectures that can learn to represent and apply edits and hope to direct the attention of the research community to this new and interesting data source, leading to better datasets and stronger models.

Briefly, the contributions of our paper are: (a) in Sect. 2, we present a new and important machine learning task on learning representations of edits (b) we present a family of models that capture the structure of edits and compute efficient representations in Sect. 3 (c) we create a new source code edit dataset, which we release at `anonymousLocation` (d) we perform a set of experiments on the learned edit representations in Sect. 4 for natural language text and source code and present promising empirical evidence that our models succeed in capturing the semantics of edits.

2 TASK

In this work, we are interested in learning to represent and apply edits on discrete sequential or structured data, such as text or source code parse trees. Figure 1 gives a graphical overview of the task, described precisely below.

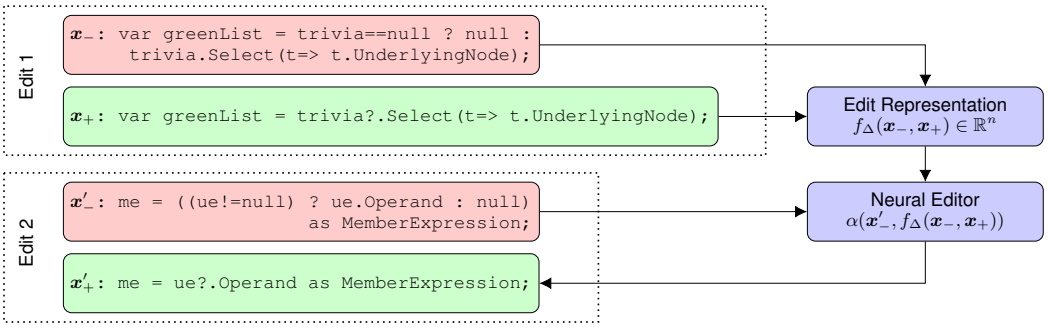

Figure 1: Given an edit (Edit 1) of $\boldsymbol{x}_-$ to $\boldsymbol{x}_+$, f_Δ computes an edit representation vector. Using that representation vector the neural editor α applies the same edit to a new $\boldsymbol{x}'_-$. The code snippets shown here are real code change examples from the `roslyn` open-source compiler project.

Edit Representation Given a dataset of edits $\{\boldsymbol{x}_-^{(i)} \rightarrow \boldsymbol{x}_+^{(i)}\}_{i=1}^N$, where $\boldsymbol{x}_-^{(i)}$ is the original version of some object and $\boldsymbol{x}_+^{(i)}$ its edited form (see upper half of Figure 1 for an example), our goal is to learn a representation function f_Δ that maps an edit operation $\boldsymbol{x}_- \rightarrow \boldsymbol{x}_+$ to a real-valued *edit representation* $f_\Delta(\boldsymbol{x}_-, \boldsymbol{x}_+) \in \mathbb{R}^n$. A desired quality of f_Δ is for the computed edit representations to have the property that semantically similar edits have nearby representations in $\mathbb{R}^n$, which allows unsupervised clustering of similar changes and many other downstream tasks.

Neural Editor Given an edit representation function f_Δ, we want to learn to apply edits in a new context. This can be achieved by learning a *neural editor* α that accepts an edit representation $f_\Delta(\boldsymbol{x}_-, \boldsymbol{x}_+)$ and a new input $\boldsymbol{x}'_-$ and generates $\boldsymbol{x}'_+$. This is illustrated in the lower half of Figure 1.

3 MODEL

We cast the edit representation problem as an autoencoding task, where we aim to minimize the reconstruction error of α for the edited version $\boldsymbol{x}_+$ given the edit representation $f_\Delta(\boldsymbol{x}_-, \boldsymbol{x}_+)$ and the original version $\boldsymbol{x}_-$. By limiting the capacity of f_Δ's output and allowing the model to freely use information about $\boldsymbol{x}_-$, we are introducing a "bottleneck" that forces the overall framework to not simply treat $f_\Delta(\boldsymbol{x}_-, \boldsymbol{x}_+)$ as an encoder of $\boldsymbol{x}_+$. The main difference from traditional autoencoders is that in our setup, an optimal solution requires to re-use as much information as possible from $\boldsymbol{x}_-$ to make the most of the capacity of f_Δ. Formally, given a probabilistic editor function P_α such as a neural network and a dataset $\{\boldsymbol{x}_-^{(i)} \rightarrow \boldsymbol{x}_+^{(i)}\}_{i=1}^N$, we seek to minimize the negative likelihood loss

$$\mathcal{L} = -\frac{1}{N} \sum_i \log P_\alpha(\boldsymbol{x}_+ \mid \boldsymbol{x}_-, f_\Delta(\boldsymbol{x}_-, \boldsymbol{x}_+)).$$

Note that this loss function can be interpreted in two ways: (1) as a conditional autoencoder that encodes the salient information of an edit, given $\boldsymbol{x}_-$ and (2) as an encoder-decoder model that encodes $\boldsymbol{x}_-$ and decodes $\boldsymbol{x}_+$ conditioned on the edit representation $f_\Delta(\boldsymbol{x}_-, \boldsymbol{x}_+)$. In the rest of this section, we discuss our methods to model P_α and f_Δ as neural networks.

3.1 NEURAL EDITOR

As discussed above, α should retrieve as much information as possible from $\boldsymbol{x}_-$, and hence, an encoder-decoder architecture with the ability to copy from the input is most appropriate. As we are primarily interested in edits on text and source code in this work, we explored two architectures: a sequence-to-sequence model for text, and a graph-to-tree model for source code, whose known semantics we can leverage both on the encoder as well as on the decoder side. Other classes of edits, for example, image manipulation, would most likely be better served by convolutional neural models.

Sequence-to-Sequence Neural Editor First, we consider a standard sequence-to-sequence model with attention (over the tokens of $\boldsymbol{x}_-$). The architecture of our sequence-to-sequence model is similar to that of Bahdanau et al. (2014), with the difference that we use a bidirectional LSTM in the encoder

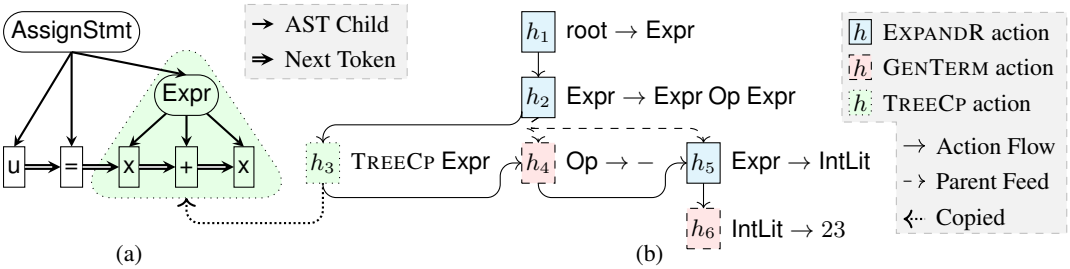

Figure 2: (a) Graph representation of statement `u = x + x`. Rectangular (resp. rounded) nodes denote tokens (resp. non-terminals). (b) Sequence of tree decoding steps yielding `x + x - 23`, where `x + x` is copied (using the TreeCp action) from the context graph in (a).

and a token-level copying mechanism (Vinyals et al., 2015) that directly copies tokens into the decoded sequence. Whereas in standard sequence-to-sequence models the decoder is initialized with the representation computed by the encoder, we initialize it with the concatenation of encoder output and the edit representation. We also feed the edit representation as input to the decoder LSTM at each decoding time step. This allows the LSTM decoder to take the edit representation into consideration while generating the output sequence.

Graph-to-Tree Neural Editor Our second model aims to take advantage of the additional structure of $\boldsymbol{x}_-$ and $\boldsymbol{x}_+$. To achieve this, we combine a graph-based encoder with a tree-based decoder. We use $T(\boldsymbol{x})$ to denote a tree representation of an element, *e.g.*, the abstract syntax tree (AST) of a fragment of source code. We extend $T(\boldsymbol{x})$ into a graph form $G(\boldsymbol{x})$ by encoding additional relationships (*e.g.*, the "next token" relationship between terminal nodes, *etc.*) (see Figure 2(a)). To encode the elements of $G(\boldsymbol{x}_-)$ into vector representations, we use a gated graph neural network (GGNN) (Li et al., 2015). Similarly to recurrent neural networks for sequences (such as biRNNs), GGNNs compute a representation for each node in the graph, which can be used in the attention mechanisms of a decoder. Additionally, we use them to obtain a representation of the full input $\boldsymbol{x}_-$, by computing their weighted average following the strategy of Gilmer et al. (2017) (*i.e.*, computing a score for each node, normalizing scores with a softmax, and using the resulting values as weights).

Our tree decoder follows the semantic parsing model of Yin & Neubig (2017), who sequentially generate a tree $T(\boldsymbol{x}_+)$ as a series of expansion actions $a_1 \dots a_N$. The probability of taking an action is modeled as $p(a_t \mid a_{<t}, s)$, where s is the input (a sequence of words in the original semantic parsing setting) and $a_{<t}$ is the partial tree that has been generated so far. The model of Yin & Neubig (2017) has two types of actions: ExpandR expands the current non-terminal using a grammar rule, and GenTerm generates a terminal token from a vocabulary or copies a token from s. The dependence on the partial tree $a_{<t}$ is modeled by an LSTM cell which is used to maintain state throughout the generation procedure. Additionally, the LSTM receives the decoder state used to pick the action at the parent node as an additional input ("parent-feeding"). This process illustrated in Figure 2(b).

We extend this model to our setting by replacing the input sequence s by $\boldsymbol{x}_-$; concretely, we condition the decoder on the graph-level representation computed for $G(\boldsymbol{x}_-)$. Additionally, we use the change representation $f_\Delta(\cdot)$ as an additional input to the LSTM initial state and at every decoding step. Based on the observation that edits to source code often manipulate the syntax tree by moving expressions around (*e.g.* by nesting statements in a conditional, or renaming a function while keeping its arguments), we extend the decoding model of Yin & Neubig (2017) by adding a facility to copy entire subtrees from the input. For this, we add a decoder action TreeCp. This action is similar to standard copying mechanism known from pointer networks (Vinyals et al., 2015), but instead of copying only a single token, it copies the whole subtree pointed to.

However, adding the TreeCp action means that there are many correct generation sequences for a target tree. This problem appears in token-copying as well, but can be easily circumvented by marginalizing over all correct choices at each generation step (by normalizing the probability distribution over allowed actions to sum up those that have the same effect). In the subtree-copying setting, this solution is insufficient, as the lengths of action sequences representing different choices may differ. We follow Liu et al. (2018) to handle this problem during training and simply pick one correct generation sequence (the one greedily selecting TreeCp) but change the objective such that

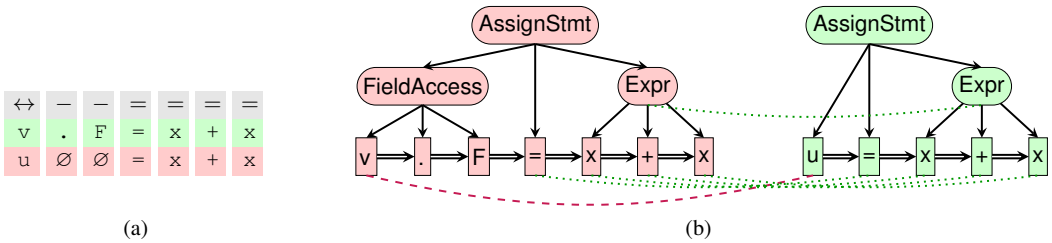

Figure 3: Sequence (a) and graph (b) representation of edit of `v.F = x + x` to `u = x + x`.

no correct decoder action choice is penalized; achieved by a "many-hot" encoding of correct choices. At test time, we resolve the issue by using beam search and merging beams with identical results.

3.2 Edit Representation

To compute a useful edit representation, a model needs to focus on the differences between $\boldsymbol{x}_-$ and $\boldsymbol{x}_+$. A risk in our framework is that f_Δ degenerates into an encoder for $\boldsymbol{x}_+$, turning α into a decoder. To avoid this, we need to follow the standard autoencoder trick, *i.e.* it is important to limit the capacity of the result of f_Δ by generating the edit representation into a low-dimensional space $\mathbb{R}^N$ that acts as a bottleneck and encodes only the information that is needed to reconstruct $\boldsymbol{x}_+$ from $\boldsymbol{x}_-$. We again experimented with both sequence-based and graph-based representations of edits.

Sequence Encoding of Edits Given $\boldsymbol{x}_-$ (resp. $\boldsymbol{x}_+$) as sequence of tokens $t_-^{(0)}, \dots t_-^{(T_-)}$ (resp. $t_+^{(0)}, \dots t_+^{(T_+)}$), we can use a standard (deterministic) diffing algorithm to compute an alignment of tokens in the two sequences. We then use extra symbols $\varnothing$ for padding, $+$ for additions, $-$ for deletions, $\leftrightarrow$ for replacements, and $=$ for unchanged tokens to generate a single sequence representing both $\boldsymbol{x}_-$ and $\boldsymbol{x}_+$. This is illustrated in Figure 3(a). By embedding the three entries in each element of the sequence separately and concatenating their representation, they can be fed into a standard sequence encoder whose final state is our desired edit representation. In this work, we use a biLSTM.

Graph Encoding of Edits As in the graph-to-tree neural editor, we represent $\boldsymbol{x}_-$ and $\boldsymbol{x}_+$ as trees $T(\boldsymbol{x}_-)$ and $T(\boldsymbol{x}_+)$. We combine these trees into a graph representation $G(\boldsymbol{x}_- \to \boldsymbol{x}_+)$ by merging both trees into one graph, using "Removed", "Added" and "Replaced" edges. To connect the two trees, we compute the same alignment as in the sequence case, connecting leaves that are the same and each replaced leaf to its replacement. We also propagate this information up in the trees, *i.e.*, two inner nodes are connected by "=" edges if all their descendants are connected by "=" edges. This is illustrated in Figure 3(b). Finally, we also use the same "+" / "-" / "↔" / "=" tags for the initial node representation, computing it as the concatenation of the string label (*i.e.* token or nonterminal name) and the embedding of the tag. To obtain an edit representation, we use a GGNN unrolled for a fixed number of timesteps and again use the weighted averaging strategy of Gilmer et al. (2017).

4 Evaluation

Evaluating an unsupervised representation learning method is challenging, especially for a newly defined task. Here, we aim to evaluate the quality of the learned edit representations with a series of qualitative and quantitative metrics on natural language and source code. Throughout the evaluation we use a fixed size of 512 for all edit representations.

4.1 Datasets

Natural Language Edits We use the **WikiAtomicEdits** (Faruqui et al., 2018) dataset of pairs of short edits on Wikipedia articles. We sampled $1040K$ edits from the English *insertion* portion of the dataset and split the samples into $1000K$/$20K$/$20K$ train-valid-test sets.

Source Code Edits To obtain a dataset for source code, we clone a set of 54 C# projects on GitHub and collected a **GitHubEdits** dataset (see Appendix A for more information). We selected all changes

in the projects that are no more than 3 lines long and whose surrounding 3 lines of code before and after the edited lines has *not* been changed, ensuring that the edits are separate and short. We then parsed the two versions of the source code and take as $\boldsymbol{x}_-$ and $\boldsymbol{x}_+$ the code that belongs to the top-most AST node that contains the edited lines. Finally, we remove trivial changes such variable renaming, changes within comments or formatting changes. Overall, this yields 111 724 edit samples. For each edit we run a simple C# analysis to detect all variables and normalize variable names such that each unique variable within $\boldsymbol{x}_-$ and $\boldsymbol{x}_+$ has a unique normalized name `V0`, `V1`, *etc.* This step is necessary to avoid the sparsity of data induced by the variety of different identifier naming schemes. We split the dataset into 91 372 / 10 176 / 10 176 samples as train/valid/test sets.

Additionally, we introduce a labeled dataset of source code edits by using C# "fixers". Fixers are small tools built on top of the C# compiler, used to perform common refactoring and modernization tasks (*e.g.*, using new syntactic sugar). We selected 16 of these fixers and ran them on 6 C# projects to generate a small **C#Fixers** dataset of 2,878 edit pairs with known semantics. We present descriptions and examples of each fixer in Appendix A.

4.2 Quality of Edit Representations

First, study the ability of our models to encode edits in a semantically meaningful way.

Visualizing Edits on Fixers Data In a first experiment, we train our sequential neural editor model on our GitHubEdits data and then compute representations for the edits generated by the C# fixers. A t-SNE visualization (Maaten & Hinton, 2008) of the encodings is shown in Figure 4. For this visualization, we randomly selected 100 examples from the edits of each fixer (if that fixer has more than 100 samples) and discarded fixer categories with less than 40 examples. Readers are referred to Appendix A for detailed descriptions of each fixer category. We find that our model produces dense clusters for simple or distinctive code edits, *e.g.* fixer `RCS1089` (using the `++` or `--` unary operators instead of a binary operator (*e.g.*, `i = i + 1` → `i++`), and fixer `CA2007` (adding `.ConfigureAwait(false)` for `await` statements). We also analyzed cases where (1) the edit examples from the same fixer are scattered, or (2) the clusters of different fixers overlap with each other. For example, the fixer `RCS1077` covers 13 different aspects of optimizing LINQ method calls (*e.g.*, type casting, counting, etc.), and hence its edits are scattered. On the other hand, fixers `RCS1146` and `RCS1206` yield overlapping clusters, as both fixers change code to use the `?.` operator. Fixers `RCS1207` (change a lambda to a method group, *e.g.* `foo(x=>bar(x))` → `foo(bar)`) and `RCS1021` (simplify lambda expressions, *e.g.* `foo(x=>{return 4;})` → `foo(x=>4)`) are similar, as both inline lambda expressions in two different ways. Analysis yields that the representation is highly dependent on surface tokens. For instance, `IDE004` (removing redundant type casts, *e.g.* `(int)2` → `2`) and `RCS1207` (removing explicit argument lists) yield overlapping clusters, as both involve deleting identifiers wrapped by parentheses.

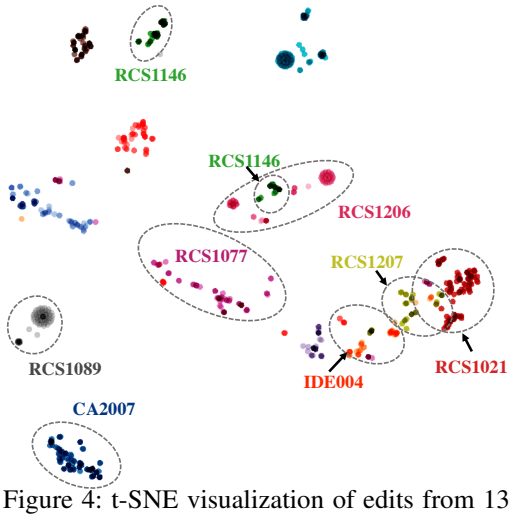

Figure 4: t-SNE visualization of edits from 13 C# fixers, where point color indicates the fixer. Labels indicate the id of the fixer, see main text.

Human Evaluation on Encoding Natural Language WikiAtomicEdits In a second experiment, we evaluate how well neighborhoods in edit representation space correspond to semantic similarity. For this, we computed the five nearest neighbors of 200 randomly sampled edits from our training set, using both our trained sequence-to-sequence editing model with sequential edit encoder, as well as a simple bag-of-words baseline based on TF-IDF scores. We then rated the quality of the retrieved neighbors on a scale of 0 ("unrelated edit"), 1 ("similar edit") and 2 ("semantically or syntactically same edit"). We show the (normalized) discounted cumulative gain (DCG, Manning et al. (2008)) for the two models at the top of Tab. 1 (higher is better). The results indicate that our neural model

	Bag of Words Model			Seq2Seq – Seq Edit Encoder		
DCG@5	9.3			13.5		
by edit size	1: 14.7	2-3: 10.8	>3: 5.4	1: 16.2	2-3: 12.9	>3: 12.4
NDCG@5	67.3%			90.3%		

she , along with her follow artist carolyn mase studied with ►*impressionist landscape painter*◄ john henry twachtman at the art students league of new york .

NN-1	his brother was draughtsman william daniell and his uncle was ►*landscape painter*◄ thomas daniell .	the first painting was a portrait of a young girl , emerantia van beresteyn , the sister of ►*the landscape painter*◄ nicolaes van beresteyn , the later founder of half of this hofje .
NN-2	william james linton (december 7 , 1812 - december 29 , 1897) was an english - born american wood engraver , ►*landscape painter ,*◄ political reformer and author of memoirs , novels , poetry and non-fiction .	he was the club 's top scorer with 22 goals in all competitions , one more than ►*senegalese striker*◄ lamine diarra , who left the club at the end of the season .
NN-3	early on , hopper modeled his style after chase and french ►*impressionist*◄ masters douard manet and edgar degas .	caforio " aggressively attacked " his opponent , ►*republican incumbent*◄ steve knight , for his delayed response to the leak .

►*daniel james nava (born february 22 , 1983) is an american professional baseball outfielder*◄ nava is only the fourth player in mlb history to hit a grand slam in his first major league at bat and the second to do it on the first pitch .

NN-1	he batted .302 with 73 steals , and received a september call - up to the major leagues ►*as an outfielder*◄ .	►*arthur ray briles (born december 3 , 1955) is a former american football coach and*◄ his most recent head coaching position was at baylor university , a position he held from the 2008 season through the 2015 season .
NN-2	he played ►*as an outfielder*◄ for the hanshin tigers .	►*jonathan david disalvatore (born march 30 , 1981) is a professional ice hockey*◄ he was selected by the san jose sharks in the 4th round (104th overall) of the 2000 nhl entry draft .
NN-3	in 2012 , his senior at oak mountain , dahl had a .412 batting average , 34 runs batted in (rbis) , and 18 stolen bases ►*as an outfielder .*◄	►*professor paul talalay (born march 31 , 1923) is the john jacob abel*◄ distinguished service professor of pharmacology and director of the laboratory for molecular sciences at johns hopkins school of medicine in baltimore .

Table 1: Natural language human evaluation results and 3 nearest neighbors. ►*Inserted text*◄ marked.

clearly outperforms the simplistic baseline. Tab. 1 also presents example edits we sampled along with their nearest neighbors, which show that the neural model succeeded in learning semantics of edits for both n-grams (upper example) and complex clauses (lower example). We observed that the edit representations learned by the neural editing model on WikiAtomicEdits are somewhat sensitive to position, *i.e.* the position of the inserted tokens in both the seed edit and the nearest neighbors is similar. This is illustrated in the first example in Tab. 1, where the second (*"senegalese striker"*) and the third (*"republican incumbent"*) nearest neighbors returned by the neural model have similar editing positions as the seed edit, while they are semantically diverse.

4.3 Edit Encoder Performance

To evaluate the performance of our two edit encoders discussed in Sect. 3.2 and disentangle it from the choice of neural editor, we train various combinations of our neural editor model and manually evaluate the quality of the edit representation. More specifically, we trained our neural editor models on GitHubEdits and randomly sampled 200 seed edits and computed their 3 nearest neighbors using each end-to-end model. We then rated the resulting groups using the same 0-2 scale as above. The resulting relevance scores are shown in Tab. 2.

Comparing the sequential edit encoders trained with Seq2Seq and Graph2Tree editors, we found that the edit encoder trained with the Graph2Tree objective performs better. We hypothesize that this is because the Graph2Tree editor better captures structural-level information about an edit. For instance, Example 1 in Tab. 3 removes explicit type casting. The Seq2Seq editor has difficulty distinguishing

Table 2: Relevance scores of human evaluation on GitHubEdits data. Acc.@1 denotes the ratio that the 1-nearest neighbor has a score 2.

Model	**DCG@3**	**NDCG@3** (%)	**Acc.@1** (%)
BoW	7.77	75.99	58.46
Seq2Seq – Seq Edit Encoder	10.09	90.05	75.90
Graph2Tree – Seq Edit Encoder	**10.56**	**91.40**	**79.49**
Graph2Tree – Graph Edit Encoder	9.44	86.20	72.31

Table 3: Two example source code edits and their nearest neighbors based on the edit representations computed by each model.

Example 1	Example 2
$\boldsymbol{x}_-$: `V0.SendSelectSoundRequest((int)V1);` $\boldsymbol{x}_+$: `V0.SendSelectSoundRequest(V1);`	$\boldsymbol{x}_-$: `string V0; string V1;` $\boldsymbol{x}_+$: `string V1; string V0;`
Seq2Seq – Seq Edit Encoder	Seq2Seq – Seq Edit Encoder
▶ $\boldsymbol{x}_-$: `V0.Debug(() => LITERAL);` $\boldsymbol{x}_+$: `V0.Debug(LITERAL);`	▶ $\boldsymbol{x}_-$: `RetryConfig V0; string V1;` $\boldsymbol{x}_+$: `string V1; RetryConfig V0;`
▶ $\boldsymbol{x}_-$: `V0.Debug(() => LITERAL);` $\boldsymbol{x}_+$: `V0.Debug(LITERAL);`	▶ $\boldsymbol{x}_-$: `string[] V0; string[] V1; int V2;` $\boldsymbol{x}_+$: `int V2; string[] V0; string[] V1;`
▶ $\boldsymbol{x}_-$: `V0.WriteCompressedInteger((uint)V1);` $\boldsymbol{x}_+$: `V0.WriteCompressedInteger(V1);`	▶ $\boldsymbol{x}_-$: `Type V0= null; BindingFlags V1= 0;` $\boldsymbol{x}_+$: `BindingFlags V1= 0; Type V0= null;`
Graph2Tree – Seq Edit Encoder	Graph2Tree – Seq Edit Encoder
▶ $\boldsymbol{x}_-$: `V0.WriteCompressedInteger((uint)V1);` $\boldsymbol{x}_+$: `V0.WriteCompressedInteger(V1);`	▶ $\boldsymbol{x}_-$: `RetryConfig V0; string V1;` $\boldsymbol{x}_+$: `string V1; RetryConfig V0;`
▶ $\boldsymbol{x}_-$: `V0.WriteCompressedInteger((uint)V1);` $\boldsymbol{x}_+$: `V0.WriteCompressedInteger(V1);`	▶ $\boldsymbol{x}_-$: `string[] V0; string[] V1; int V2;` $\boldsymbol{x}_+$: `int V2; string[] V0; string[] V1;`
▶ $\boldsymbol{x}_-$: `V0.WriteCompressedInteger((uint)V1);` $\boldsymbol{x}_+$: `V0.WriteCompressedInteger(V1);`	▶ $\boldsymbol{x}_-$: `int V0 = V1; int V2 = V3;` $\boldsymbol{x}_+$: `int V2 = V3; int V0 = V1;`
Graph2Tree – Graph Edit Encoder	Graph2Tree – Graph Edit Encoder
▶ $\boldsymbol{x}_-$: `V0.UpdateLastRead(this.V1);` $\boldsymbol{x}_+$: `V0.UpdateLastRead(V1);`	▶ $\boldsymbol{x}_-$: `RetryConfig V0; string V1;` $\boldsymbol{x}_+$: `string V1; RetryConfig V0;`
▶ $\boldsymbol{x}_-$: `V0.UpdateLastWrite(this.V1);` $\boldsymbol{x}_+$: `V0.UpdateLastWrite(V1);`	▶ $\boldsymbol{x}_-$: `int V0 = V1; int V2 = V3;` $\boldsymbol{x}_+$: `int V2 = V3; int V0 = V1;`
▶ $\boldsymbol{x}_-$: `V0.Append(this.V1);` $\boldsymbol{x}_+$: `V0.Append(V1);`	▶ $\boldsymbol{x}_-$: `double V0= -1; double V1= -1;` $\boldsymbol{x}_+$: `double V1= -1; double V0= -1;`

this type of edit, confusing it with changes of lambda expressions to method groups (1st and 2nd nearest neighbors) since both two types of edits involve removing paired parentheses.

Surprisingly, we found that the graph-based edit encoder does not outperform the sequence-based encoder. However, we observe that the graph edit encoder in many cases tends to better capture high-level and abstract structural edit patterns. Example 2 in Tab. 3 showcases a seed edit that swaps two consecutive declarations, which corresponds to swapping the intermediate `Expression` nodes representing each statement on the underlying AST. In this case, the graph edit encoder is capable of grouping semantically similar edits, while it seems to be more difficult for the sequential encoder encoder to capture the edit pattern. On the other hand, we found that the graph edit encoder often fails to capture simpler, lexical level edits (*e.g.*, Example 1). This might suggest that terminal node information is not effectively propagated, an interesting issue worth future investigation.

4.4 Precision of Neural Editors

Finally, we evaluate the performance of our end-to-end system by predicting the edited input $\boldsymbol{x}_+$ given $\boldsymbol{x}_-$ and the edit representation. We are interested in answering two research questions: *First*, how well can our neural editors generate $\boldsymbol{x}_+$ given the gold-standard edit representation $f_\Delta(\boldsymbol{x}_-, \boldsymbol{x}_+)$? *Second*, and perhaps more interestingly, can we use the representation of a similar edit $f_\Delta(\boldsymbol{x}'_-, \boldsymbol{x}'_+)$ to generate $\boldsymbol{x}_+$ by *applying* that edit to $\boldsymbol{x}_-$ (*i.e.* $\hat{\boldsymbol{x}}_+ = \alpha(\boldsymbol{x}_-, f_\Delta(\boldsymbol{x}'_-, \boldsymbol{x}'_+))$)?

To answer the first question, we trained our neural editor models on the WikiAtomicEdits and the GitHubEdits dataset, and evaluate the performance of encoding and applying edits on test sets. Tab. 4 lists the evaluation results. On WikiAtomicEdits, our Seq2Seq editor with the sequential edit encoder achieves an accuracy of 72.9%, demonstrating that the neural editor is capable of using the edit

Table 4: Test performance of different neural editors.

Model	**Acc.@1** (%)	**Recall@5** (%)	**PPL** per token
GitHubEdits			
Seq2Seq – Seq Edit Encoder	**59.63**	**65.46**	**1.2792**
Graph2Tree – Seq Edit Encoder	57.49	62.94	1.3043
Graph2Tree – Graph Edit Encoder	48.05	56.51	1.3712
WikiAtomicEdits			
Seq2Seq – Seq Edit Encoder	72.94	76.53	1.0527

Table 5: Transfer learning results on C# fixers data, averaged across all fixer categories.

Model	**Acc.**(%)	**Acc.***(%)	**Recall@5**(%)	**Recall@5***(%)
Seq2Seq – Seq Edit Encoder	38.35	**77.67**	41.50	**83.84**
Graph2Tree – Seq Edit Encoder	**49.21**	77.30	**51.93**	81.77

*: upper-bound performance of predicting $\boldsymbol{x}_+$ using the gold-standard edit representations.

representation to make high-quality edits. On the GitHubEdits dataset, we find that the Seq2Seq editor with sequential edit encoder registers the best performance. However, it should be noted that better performance does *not* necessarily imply better (more generalizable) edit representation, since we encode the gold-standard edit $f_\Delta(\boldsymbol{x}_-, \boldsymbol{x}_+)$ to predict $\boldsymbol{x}_+$. Nevertheless, we hypothesize that the higher accuracy of the Seq2Seq edit is due to the fact that a significant proportion of edits in this dataset is small and primarily syntactically simple. Indeed we find that 69% of test examples have a token-level edit distance of less than 5.

To answer the second question, we use the trained neural editors from the previous experiment, and test their performance in an "one-shot" transfer learning scenario. Specifically, we use our high-quality C#Fixers dataset, and for each fixer category $\mathcal{F}$ of semantically similar edits, we randomly select a seed edit $\{\boldsymbol{x}'_- \rightarrow \boldsymbol{x}'_+\} \in \mathcal{F}$, and use its edit representation $f_\Delta(\boldsymbol{x}'_-, \boldsymbol{x}'_+)$ to predict the updated code for *all* examples in $\mathcal{F}$, *i.e.*, we have $\hat{\boldsymbol{x}}_+ = \alpha(\boldsymbol{x}_-, f_\Delta(\boldsymbol{x}'_-, \boldsymbol{x}'_+)), \forall\ \{\boldsymbol{x}_- \rightarrow \boldsymbol{x}_+\} \in \mathcal{F}$. This task is highly non-trivial, since a fixer category could contain more than hundreds of edit examples collected from different C# projects. Therefore, it requires the edit representations to generalize and transfer well, while being invariant of local lexical information like specific method names. To make the experimental evaluation more robust to noise, for each fixer category $\mathcal{F}$, we randomly sample 10 seed edit pairs $\{\boldsymbol{x}'_- \rightarrow \boldsymbol{x}'_+\}$, compute their edit representations and use them to predict the edited version of the examples in $\mathcal{F}$ and evaluate accuracy of predicting the exact final version. We then report the best score among the 10 seed representations as the performance metric on $\mathcal{F}$.

Tab. 5 summarizes the results and also reports the upper bound performance when using the gold-standard edit representation $f_\Delta(\boldsymbol{x}_-, \boldsymbol{x}_+)$ to predict $\boldsymbol{x}_+$. We found that our neural Graph2Tree editor with the sequential edit encoder significantly outperforms the Seq2Seq editor, even though Seq2Seq performs better when using gold-standard edit representations. This suggest that the edit representations learned with the Graph2Tree model generalize better, especially for edits discussed in Sect. 4.2 that involve syntactic variations like `RCS1021` (lambda expression simplification, 7.8% *vs.* 30.7% for Seq2Seq and Graph2Tree), and `RCS1207` (change lambdas to method groups, 7.1% *vs.* 26.2%). Interestingly, we also observe that Seq2Seq outperforms the Graph2Tree model for edits with trivial surface edit sequences, where the Graph2Tree model does not have a clear advantage. For example, on `RCS1015` (use `nameof` operator, *e.g.* `Exception("x")` $\rightarrow$ `Exception(nameof(x))`), the accuracies for Seq2Seq and Graph2Tree are 40.0% (14/35) and 28.6% (10/35), resp. We include more statistics of the results in Appendix C.

5 Related Work

Edits have recently been considered in NLP by Faruqui et al. (2018), as they represent interesting linguistic phenomena in language modeling and discourse. Related to this work, Guu et al. (2017) present a generative model of natural language sentences via editing prototypes. In contrast to these examples of natural language generative modeling, our work focuses on representing edits in a semantically meaningful way. We are not aware of any related work that classifies or otherwise represents the differences over similar input, with the exception of differential recurrent neural networks used

for action recognition in videos (Veeriah et al., 2015; Zhuang et al., 2018). This is a substantially different task, as the data includes a temporal component as well.

Source code edits are a widely studied artifact. Specialized software, such as git, is widely used to store source code revision histories. Nguyen et al. (2013) studied the repetitiveness of source code changes by identifying identical types of changes using a deterministic differencing tool. In contrast, we employ on a neural network to cluster *similar* changes together. Rolim et al. (2017) use such clusters to synthesize small programs that perform the edit. The approach is based on Rolim et al. (2018) extract manually designed syntactic features from code and cluster over multiple changes to find repeatable edit rules. Similarly, Paletov et al. (2018) extract syntactic features specifically targeting edits in cryptography API protocols. In this work, we try to avoid hand-designed features and allow a neural network to learn the relevant aspects of a change by directly giving as input the original and final version of a changed code snippet.

Modeling tree generation with machine learning is an old problem that has been widely studied in NLP. Starting with Maddison & Tarlow (2014), code generation has also been considered as a tree generation problem. Close to our work is the decoder of Yin & Neubig (2017) which we use as the basis of our decoder. The work of Chen et al. (2018) is also related, since it provides a tree-to-tree model, but focuses on learning a single translation tasks and cannot be used directly to represent multiple types of edits. Both Yin & Neubig (2017) and Chen et al. (2018) have copying mechanism for single tokens, but our subtree copying mechanism is novel.

Autoencoders (see Goodfellow et al. (2016) for an overview) have a long history in machine learning. Variational autoencoders (Kingma & Welling, 2013) are similar to autoencoders but instead of focusing on the learned representation, they aim to create accurate generative probabilistic models. Most (variational) autoencoders focus on encoding images but there have been works that autoencode sequences, such as text (Dai & Le, 2015; Yang et al., 2017; Bowman et al., 2015) and graphs (Simonovsky & Komodakis, 2018; Liu et al., 2018). Conditional variational autoencoders (Sohn et al., 2015) have a related form to our model (with the exception of the KL term), but are studied as generative models, whereas we are primarily interested in the edit representation.

6 Discussion & Conclusions

In this work, we presented the problem of learning distributed representation of edits. We believe that the dataset of edits is highly relevant and should be studied in more detail. While we have presented a set of initial models and metrics on the problem and obtained some first promising results, further development in both of these areas is needed. We hope that our work inspires others to work on this interesting problem in the future.

References

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

Samuel R Bowman, Luke Vilnis, Oriol Vinyals, Andrew M Dai, Rafal Jozefowicz, and Samy Bengio. Generating sentences from a continuous space. *arXiv preprint arXiv:1511.06349*, 2015.

Xinyun Chen, Chang Liu, and Dawn Song. Tree-to-tree neural networks for program translation. *arXiv preprint arXiv:1802.03691*, 2018.

Andrew M Dai and Quoc V Le. Semi-supervised sequence learning. In *Advances in neural information processing systems*, pp. 3079–3087, 2015.

Manaal Faruqui, Ellie Pavlick, Ian Tenney, and Dipanjan Das. WikiAtomicEdits: A multilingual corpus of Wikipedia edits for modeling language and discourse. In *EMNLP*, 2018.

Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *International Conference on Machine Learning (ICML)*, 2017.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

Kelvin Guu, Tatsunori B Hashimoto, Yonatan Oren, and Percy Liang. Generating sentences by editing prototypes. *arXiv preprint arXiv:1709.08878*, 2017.

Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

Qi Liu, Miltiadis Allamanis, Marc Brockschmidt, and Alexander L. Gaunt. Constrained graph variational autoencoders for molecule design. In *Neural Information Processing Systems (NIPS)*, 2018.

Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-SNE. *Journal of machine learning research*, 9(Nov):2579–2605, 2008.

Chris Maddison and Daniel Tarlow. Structured generative models of natural source code. In *International Conference on Machine Learning (ICML)*, 2014.

Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008. ISBN 0521865719, 9780521865715.

Hoan Anh Nguyen, Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N Nguyen, and Hridesh Rajan. A study of repetitiveness of code changes in software evolution. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pp. 180–190. IEEE Press, 2013.

Rumen Paletov, Petar Tsankov, Veselin Raychev, and Martin Vechev. Inferring crypto API rules from code changes. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 450–464. ACM, 2018.

Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. Learning syntactic program transformations from examples. In *Proceedings of the 39th International Conference on Software Engineering*, pp. 404–415. IEEE Press, 2017.

Reudismam Rolim, Gustavo Soares, Rohit Gheyi, and Loris D'Antoni. Learning quick fixes from code repositories. *arXiv preprint arXiv:1803.03806*, 2018.

Martin Simonovsky and Nikos Komodakis. GraphVAE: Towards generation of small graphs using variational autoencoders. *arXiv preprint arXiv:1802.03480*, 2018.

Kihyuk Sohn, Honglak Lee, and Xinchen Yan. Learning structured output representation using deep conditional generative models. In *Advances in Neural Information Processing Systems*, pp. 3483–3491, 2015.

Vivek Veeriah, Naifan Zhuang, and Guo-Jun Qi. Differential recurrent neural networks for action recognition. In *Proceedings of the IEEE international conference on computer vision*, pp. 4041–4049, 2015.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *Advances in Neural Information Processing Systems*, pp. 2692–2700, 2015.

Zichao Yang, Zhiting Hu, Ruslan Salakhutdinov, and Taylor Berg-Kirkpatrick. Improved variational autoencoders for text modeling using dilated convolutions. *arXiv preprint arXiv:1702.08139*, 2017.

Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pp. 440–450, 2017.

Naifan Zhuang, The Duc Kieu, Guo-Jun Qi, and Kien A Hua. Deep differential recurrent neural networks. *arXiv preprint arXiv:1804.04192*, 2018.

A DATASETS AND CONFIGURATION

WikiAtomicEdits We randomly sampled $1040K$ insertion examples from the English portion of WikiAtomicEdits (Faruqui et al., 2018) dataset, with a train, dev and test splits of $1000K$, $20K$ and $20K$.

GitHubEdits We cloned the top 54 C# GitHub repositories based on their popularity (Tab. 8). For each commit in the `master` branch, we collect the previous and updated versions of the source code, and extract all consecutive lines of edits that are smaller than three lines, and with at least three preceding and successive lines that have not been changed. We then filter trivial changes such as variable and identifier renaming, and changes happened within comments. We also limit the number of tokens for each edit to be smaller than 100, and down-sample edits whose frequency is larger than 30. Finally, we split the dataset by commit ids, ensuring that there are no edits in the training and testing (development) sets coming from the same commit. Tab. 6 lists some statistics of the dataset.

Table 6: Statistics of the GitHubEdits Dataset

Average Num. Tokens in $\boldsymbol{x}_-$	16.4
Average Num. Tokens in $\boldsymbol{x}_+$	17.0
Average Edit Distance	5.0
Average size of AST for $T(\boldsymbol{x}_-)$	28.5
Average size of AST for $T(\boldsymbol{x}_+)$	29.4

C#Fixers We selected 16 C# fixers from Roslyn[1] and Roslynator[2], and ran them on 6 C# projects to generate a small, high-quality C# fixers dataset of 2 878 edit pairs with known semantics. Table 7 lists the detailed descriptions for each fixer category. And more information can be found at `https://github.com/JosefPihrt/Roslynator/blob/master/src/Analyzers/README.md`.

Network Configuration Throughout the experiments, we use a fixed edit representation size of 512. The dimensionality of word embedding, the hidden states of the encoder LSTMs, as well as the gated graph neural network is 128, while the decoder LSTM uses a larger hidden size of 256. For the graph-based edit encoder, we used a two-layer graph neural network, with 5 information propagation steps at each layer. During training, we performed early stopping, and choose the best model based on perplexity scores on development set. During testing, we decode a target element $\boldsymbol{x}_+$ using beam size with a beam size of 5.

B CLUSTERING EXPERIMENTS

To qualitatively evaluate the quality of the learned edit representations. We use the models trained on the WikiAtomicEdits and GitHubEdits datasets to cluster natural language and code edits. We run K-Means clustering algorithm on 0.5 million sampled edits from WikiAtomicEdits, and all $90K$ code edits fromGitHubEdits, producing 50 000 and 20 000 clusters for each dataset.

Tab. 9 and Tab. 10 list some example clusters on WikiAtomicEdits and GitHub datasets, respectively. Due to the size of clusters, we omit out-liners and present distinctive examples from each cluster. On the WikiAtomicEdits dataset, we found clusters whose examples are semantically and syntactically similar. More interestingly, on the source code data, we find representative clusters that relate to idiomatic patterns and best practices of programming. The clustering results produced by our model would be useful for programming synthesis toolkits to generate interpretable code refractory rules, which we leave as interesting future work.

[1] `http://roslyn.io`

[2] `https://github.com/JosefPihrt/Roslynator`

Table 7: Descriptions of fixer categories in C#Fixers dataset

Fixer ID	Description	Num. Edits	Example
CA2007	apply `.ConfigureAwait(false)` to `await` statements	1051	$\boldsymbol{x}_-$: `await Console.WriteAsync()` $\boldsymbol{x}_+$: `await Console.WriteAsync() .ConfigureAwait(false)`
IDE0004	Cast is redundant	53	$\boldsymbol{x}_-$: `var x = 1; var b = (int)x;` $\boldsymbol{x}_+$: `var x = 1; var b = x;`
RCS1015	Use `nameof` operator	35	$\boldsymbol{x}_-$: `Exception("parameter");` $\boldsymbol{x}_+$: `Exception(nameof(parameter));`
RCS1021	Simplify lambda expression	411	$\boldsymbol{x}_-$: `var x = items.Select(f => { return f.ToString(); });` $\boldsymbol{x}_+$: `var x = items.Select( f => f.ToString());`
RCS1032	Remove redundant parentheses	24	$\boldsymbol{x}_-$: `if ((x)) {}` $\boldsymbol{x}_-$: `if (x) {}`
RCS1058	Use compound assignment	43	$\boldsymbol{x}_-$: `i = i + 2;` $\boldsymbol{x}_+$: `i += 2;`
RCS1077	Optimize LINQ method call	200	$\boldsymbol{x}_-$: `items.Where(f => Foo(f)).Any();` $\boldsymbol{x}_+$: `items.Any(f => Foo(f));`
RCS1089	Use `--`/`++` operator instead of assignment	75	$\boldsymbol{x}_-$: `i = i + 1;` $\boldsymbol{x}_+$: `i += 1;`
RCS1097	Remove redundant `ToString` call	20	$\boldsymbol{x}_-$: `var x = s.ToString();` $\boldsymbol{x}_+$: `var x = s;`
RCS1118	Mark local variable as const	477	$\boldsymbol{x}_-$: `string s = "a"; string s2 = s + "b";` $\boldsymbol{x}_+$: `const string s = "a"; string s2 = s + "b";`
RCS1123	Add parentheses according to operator precedence	109	$\boldsymbol{x}_-$: `if (x || y && z) {}` $\boldsymbol{x}_+$: `if (x || (y && z) ) {}`
RCS1146	Use conditional access	71	$\boldsymbol{x}_-$: `x != null && x.StartsWith("a");` $\boldsymbol{x}_+$: `x?.StartsWith("a");`
RCS1197	Optimize call of `StringBuilder`'s `Append`/`AppendLine`	95	$\boldsymbol{x}_-$: `sb.Append(s + "x");` $\boldsymbol{x}_+$: `sb.Append(s).Append("x");`
RCS1202	Avoid `NullReferenceException`	56	$\boldsymbol{x}_-$: `items.First().ToString();` $\boldsymbol{x}_+$: `items?.First().ToString();`
RCS1206	Use conditional access instead of conditional expression	116	$\boldsymbol{x}_-$: `int i = (x != null) ? x.Value.GetHashCode() : 0;` $\boldsymbol{x}_+$: `int i = x?.GetHashCode() ?? 0;`
RCS1207	Use method group instead of anonymous function	42	$\boldsymbol{x}_-$: `items.Select(f => Foo(f));` $\boldsymbol{x}_+$: `items.Select(Foo);`

C Break-down Analysis of Transfer Learning Results

Tab. 11 lists the detailed evaluation results for the transfer learning experiments discussed in Sect. 4.4. The neural Graph2Tree editor outperforms the Seq2Seq editor on 10 out of 16 fixer categories. Interestingly, we found that there are categories where the end-to-end system under-performs, even though the upper-bound accuracy is high (*e.g.* `RCS1077`, `RCSRCS1197`, `RCS1207`). While these are more complex refractory rules where a single local seed edit might not be able to generalize to

Table 8: Our C# GitHub dataset projects

Name	GitHub Id	Description
acat	intel/acat	Assistive Context-Aware Toolkit
akka.net	akka/akka.net	Distributed Actors
aspnetboilerplate	aspnetboilerplate/aspnetboilerplate	ASP.NET boilerplate
AutoMapper	AutoMaper/AutoMapper	Object-Object Mapper
BotBuilder	Microsoft/BotBuilder	Bot Framework
CefSharp	cefsharp/CefSharp	Chromium Embedded Framework Bindings
choco	chocolatey/choco	package mananger
cli	dotnet/cli	.NET CLI Tools
CodeHub	CodeHubApp/CodeHub	iOS application
coreclr	dotnet/coreclr	.NET Framework
corefx	dotnet/corefx	.NET FOundational Libraries
dapper	StackExchange/Dapper	Object Mapper
dnSpy	0xd4d/dnSpy	.NET debugger and assembly editor
duplicati	duplicati/duplicati	Encrypted Cloud Backups
EntityFramework	aspnet/EntityFramework	Object-Relational Mapper
EntityFrameworkCore	aspnet/EntityFrameworkCore	Object-Relational Mapper – Core
FluentValidation	JeremySkinner/FluentValidation	Validation Rules
framework	accord-net/framework	ML, CV Framework
GVFS	Microsoft/VFSForGit	Git Virual File System
Hangfire	HangfireIO/Hangfire	Background job library
ILSpy	icsharpcode/ILSpy	Decompiler
JavaScriptServices	aspnet/JavaScriptServices	ASP.NET JS Services
MahApps.Metro	MahApps/MahApps.Metro	WPF Framework
MaterialDesignInXamlToolkit	MaterialDesignInXamlToolkit/ MaterialDesignInXamlToolkit	Design XAML & WPF
mono	mono/mono	.NET implementation
monodevelop	mono/monodevelop	IDE
MonoGame	MonoGame/MonoGame	Game Framework
msbuild	Microsoft/msbuild	Build Tool
Mvc	aspnet/Mvc	MVC Framework
Nancy	NancyFx/Nancy	HTTP based services
Newtonsoft.Json	JamesNK/Newtonsoft.Json	JSON framework
NLog	NLog/NLog	Loggin for .NET
OpenLiveWriter	OpenLiveWriter/ OpenLiveWriter	Text editor
OpenRA	OpenRA/OpenRA	Strategy Game Engine
Opserver	opserver/Opserver	Monitoring System
orleans	dotnet/orleans	Distributed Virtual Actors
PowerShell	PowerShell/PowerShell	Command Line
Psychson	brandonlw/Psychson	Firmware
PushSharp	Redth/PushSharp	Push Notifications
ravendb	ravendb/ravendb	Database
ReactiveUI	reactiveui/ReactiveUI	Reactive MVC Framework
RestSharp	restsharp/RestSharp	HTTP/REST Client
roslyn	dotnet/roslyn	.NET Compiler
Rx.NET	dotnet/reactive	Reactive extensions.
ServiceStack	ServiceStack/ServiceStack	Web Service Framework
shadowsocks-windows	shadowsocks/ shadowsocks-windows	Cryptography
ShareX	ShareX/ShareX	Screen Recorder
SignalR	SignalR/SignalR	Real-time web framework
Sonarr	Sonarr/Sonarr	PVR
SpaceEngineers	KeenSoftwareHouse/ SpaceEngineers	Game
SparkleShare	hbons/SparkleShare	File Sharing
StackExchange.Redis	StackExchange/ StackExchange.Redis	Redis Client
WaveFunctionCollapse	mxgmn/ WaveFunctionCollapse	Bitmap/tilemap Generator
Wox	Wox-launcher/Wox	Launcher

Table 9: Example clusters on WikiAtomicEdits data using representations learned by a neural Seq2Seq editor with sequential edit encoder

Description	Add a person's middle name
1.	isaiah ►*marcus*◄ rankin (born 22 may 1978 in london) is an english professional footballer currently playing for stevenage borough .
2.	audrey ►*kathleen*◄ brown (born 24 may , 1913) is a british athlete who competed mainly in the 100 metres .
3.	alice ►*edith*◄ rumph was a painter , etcher , and teacher .
4.	mark ►*larry*◄ taufua is an australian professional rugby league player .
5.	monique ►*edith*◄ lamoureux (born july 3 , 1989) is an american ice hockey player .
Description	Add a parenthetical expression *also ... as* to modify the subject
1.	mid-state regional airport ►*, also known as mid-state airport ,*◄ is a small airport on in rush township , centre county in pennsylvania in the united states .
2.	islamic culture ►*, also known as saracenic culture ,*◄ is a term primarily used in secular academia to describe the cultural practices common to historically islamic peoples .
3.	birds of prey ►*, also known as raptors ,*◄ are birds that hunt for food primarily via flight , using their keen senses , especially vision .
4.	tetyana styazhkina ►*, also written as tetyana stiajkina ,*◄ (; born april 10 , 1977) is a ukrainian cycle racer who rides for the chirio forno d'asolo team .
5.	acid jazz ►*, also known as club jazz ,*◄ is a musical genre that combines elements of jazz , soul , funk , disco and hip hop .
Description	Specify location using a prepositional phrase.
1.	the douro fully enters portuguese territory just after the confluence with the gueda river ; once the douro enters portugal , major population centres are less frequent ►*along the river*◄ .
2.	mochou lake and mochou lake park are located at 35 hanzhongmen da jie in the jianye district of nanjing , china ►*, west to qinhuai river*◄ .
3.	reiner gamma is an albedo feature that is located on the oceanus procellarum , to the west of the reiner crater ►*on the moon*◄ .
4.	she made a brief return to the screen in " parrish " (1961) , playing the supporting role of mother which received little attention ►*by the press*◄ .
5.	he was involved in a few storylines , including one where he broke his toe and had a heart attack after he was pushed by a mugger ►*in the market*◄ .
Description	Add positional or temporal clause
1.	►*at the time*◄ ajax and hercules were trapped behind a landslide at the gaillard cut , both were working to clear the landslide .
2.	►*at the docks ,*◄ hikaru attempts to befriend the tiger , but finds that it dislikes humans .
3.	►*about the second ,*◄ i do know they exist , but the question is whether they are considered a genre outside of japan .
4.	►*in the battle ,*◄ shirou uses his reality marble , unlimited blade works and defeats gilgamesh .
5.	►*in the game ,*◄ red is a curious 11 - year - old boy from pallet town .

all scenarios, improving the generalization ability of the edit representations and the performance of end-to-end system would be interesting future directions.

Table 10: Example clusters on GithubEdits data using representations learned by a Graph2Tree editor with sequential edit encoder. Locally defined variable names are canonicalized.

Description Switch from `Assert.Equal` to `Assert.Empty`

```
x−  Assert.Equal(0, V0.ProjectIds.Count);
x+  Assert.Empty(V0.ProjectIds);

x−  Assert.Equal(0, V0.ProjectReferences.Count());
x+  Assert.Empty(V0.ProjectReferences);

x−  Assert.Equal(0, V0.TrustedSelectionPaths.Count);
x+  Assert.Empty(V0.TrustedSelectionPaths);

x−  Assert.Equal(0, V0.Count);
x+  Assert.Empty(V0);

x−  Assert.Equal(0, V0.Messages.Count);
x+  Assert.Empty(V0.Messages);
```

Description Use conditional access

```
x−  Type V0 = V1 == null ? null : V1.GetType();
x+  Type V0 = V1?.GetType();

x−  V0 = ((V1!= null) ? V1.Operand : null) as MemberExpression;
x+  V0 = V1?.Operand as MemberExpression;

x−  string V0 = V1 == null ? null : V1.GetType().Name;
x+  string V0 = V1?.GetType().Name;

x−  var V0 = V1 == null ? null : V1(V2).ToArray();
x+  var V0 = V1?.Invoke(V2).ToArray();

x−  Assert.Equal(0, V0.Messages.Count);
x+  Assert.Empty(V0.Messages);
```

Description Optimize LINQ queries

```
x−  var V0 = V1.Customers.Where(V2 => V2.CustomerID == LITERAL)
                .FirstOrDefault();
x+  var V0 = V1.Customers
              .FirstOrDefault(V2 => V2.CustomerID == LITERAL);
x−  var V0 = V1.TypeConverters.Where(V2 => V2.CanConvertTo(V3, V1))
              .FirstOrDefault();
x+  var V0 = V1.TypeConverters
             .FirstOrDefault(V2 => V2.CanConvertTo(V3, V1));
x−  var V0 = this.V1.Where(V2 => V2.CanDeserialize(V3))
                 .FirstOrDefault();
x+  var V0 = this.V1.FirstOrDefault(V2 => V2.CanDeserialize(V3));
x−  var V0 = V1.Where(V2 => V2.Item1 == V3 && V2.Item2 == V4)
                .FirstOrDefault();
x+  var V0 = V1.FirstOrDefault(V2 => V2.Item1 == V3 && V2.Item2 == V4);
```

Description Change from `Add` function to indexer.

```
x−  V0.Add(V1.key, V1.V2);
x+  V0[V1.key] = V1.V2;

x−  V0.Add(V1.Id, V2);
x+  V0[V1.Id] = V2;

x−  V0.Add(V1.Etag, V1);
x+  V0[V1.Etag] = V1;

x−  V0.Add(V1.V2, V3.Merge(V1.V4));
x+  V0[V1.V2] = V3.Merge(V1.V4);
```

Table 11: Break-down performance results on the transfer learning task. See Tab. 7 for descriptions of each fixer category.

	Graph2Tree — Seq Edit Encoder				Seq2Seq — Seq Edit Encoder			
Fixer ID	**Acc.**(%)	**Acc.***(%)	**Recall@5**(%)	**Recall@5***(%)	**Acc.**(%)	**Acc.***(%)	**Recall@5**(%)	**Recall@5***(%)
CA2007	88.0	89.2	88.2	94.3	52.7	91.9	61.0	93.8
IDE0004	69.8	92.5	73.6	94.3	45.3	98.1	45.3	98.1
RCS1015	28.6	82.9	40.0	82.9	40.0	71.4	42.9	71.4
RCS1021	30.7	60.8	33.3	67.6	7.8	56.2	17.8	72.3
RCS1032	8.3	37.5	8.3	45.8	20.8	45.8	20.8	45.8
RCS1058	93.0	88.4	95.3	90.7	37.2	69.8	39.5	76.7
RCS1077	6.5	69.5	6.5	74.0	7.5	84.0	7.5	84.5
RCS1089	96.0	98.7	98.7	98.7	76.0	98.7	76.0	98.7
RCS1097	15.0	90.0	15.0	90.0	25.0	90.0	25.0	95.0
RCS1118	95.4	98.1	99.6	99.6	93.7	99.6	98.7	1.00
RCS1123	66.1	81.7	68.8	86.2	64.2	87.2	65.1	94.5
RCS1146	54.9	81.7	56.3	85.9	45.1	76.1	57.7	91.5
RCS1197	5.3	25.3	5.3	33.7	12.6	40.0	12.6	50.0
RCS1202	28.6	67.9	37.5	75.0	28.6	69.6	32.1	80.4
RCS1206	75.0	99.1	75.9	99.1	50.0	1.00	50.0	1.00
RCS1207	26.2	73.8	28.6	90.5	7.1	64.3	11.9	88.1

*: upper-bound performance of predicting $\boldsymbol{x}_+$ using the gold-standard edit representations.